V. Kůrková et al. (Eds.): ITAT 2014 with selected papers from Znalosti 2014, CEUR Workshop Proceedings Vol. 1214,
http://ceur-ws.org/Vol-1214, Series ISSN 1613-0073, © 2014 R. Mařík

Structure Composition Discovery using Alloy

Radek Mařík

The Czech Technical University (Prague), Czech Republic
marikr@fel.cvut.cz

Abstract. A method for structure discovery is described. We assume that an unknown composite structure is sampled and observed as a set of small relational structure fractions. Each fraction is described by a set of atoms and their relations. We propose a technique that allows fusing these fractions into large components or the structure itself. The method is demonstrated using an example taken from the genealogy domain. Our approach of the solution is based on SAT. We show that the composition can be effectively performed using the Alloy tool, which is a bounded model checker. The search through the possible solutions is guided by the unique identities of observations.

1 Introduction

A general scientific approach begins with a strategy of collecting of evidence events [10]. In the next step the event data are sorted and classified. We often try to find abstract entities and their attributes to simplify model building. Relations among entities are employed in the actual model composition. Based on the model we can derive or predict further properties, events, and general behavioral patterns of the inspected system.

In this paper we focus on the building of such a model **instance** which resembles all non-trivial observations made on an unknown system. We assume the existence of background knowledge that is satisfied by all system model instances and has a form of relations from which any instance can be composed. Furthermore, we assume the existence of a mapping that relates facts of observations to the model instance.

While a constraint satisfaction problem seeks a solution given a set of constraints [13], our proposed method utilizes additional conditions, i.e. the input variable values are fixed and linked through the identity of particular observations. We do not deal with object detection as these methods try to select a part of an inspected system, i.e. as points of a flat n-dimensional space in pattern recognition [4, 6, 5, 13] or as relation components, such as topics or named entities in natural language processing NLP [16, 17] or communities in social network analysis [3, 9]. While general machine learning methods [13] or methods focused on inductive logic programming [8], logical and relational learning [11] try to detect rules that are followed in the inspected system, our method assumes an explicit specification of such constraints at input.

2 Structure Composition Method

In this paper we deal with a method that enables the composition of a large structure from its small predefined general fragments so that observation relations are satisfied.

We search for a resulting structure that uses a minimum number of fragments. More specifically, the resulting structure is assumed to have the form of a weighted graph $G = (V, E)$, where $V(G)$ is a set of weighted nodes and $E(G)$ is a set of weighted edges. In this paper we consider possible weights as finite sets of labels. More formally, we have a finite set X of finite sets $Y_i \in X$ of labels $L_i^j \in Y_i$. Then, there are two sets of weight functions assigning labels to vertices $WG : V(G) \rightarrow L_i^j$ and to edges $WE : E(G) \rightarrow L_k^\ell$. Observations in the form of small graphs follow the same pattern, i.e. graphs $G_m = (V_m, E_m)$, where $V_m(G_m)$ is a set of weighted nodes and $E_m(G_m)$ is a set of weighted edges and their weight domains are taken from X.

The input data are very small subgraphs with few nodes and edges. The result structure is not known a priori. Also no unique identifiers of nodes relating to the result structure are provided. The goal is a composition of subgraphs that results in as small a result graph as possible. In other words, we try to replace a large set of small subgraphs with one larger structure. We hope that all input observed subgraphs can be mapped as substructures onto the large structure. Thus, we search for a mapping between nodes V_m of small subgraphs and nodes V of the result structure and similarly for a mapping between sequences of edges E_m of small subgraphs and sequences of edges E of the result structure.

In reality, there are a number of constraints that must be satisfied in substructures, similarly the result structure must follow a number of constraints. Further, a mapping between the observed substructures and a result structure is not given explicitly. We assume that the mapping can be a very general relation specified again by constraints. Then, we search for such result structures which satisfy their own constraints and fulfill also the constraints of the mapping.

SAT can be proposed as a suitable method for searching for a solution although we might use more efficient methods for special forms of structures, like trees, and their constraints . While traditional methods provide a huge set of independent constraints leading to very difficult conditions for searching, we exploit **the identity of the observation substructures** as the fundamental feature of our method. Each substructure produces a number of constraints, but all of them are bound to the identity of a given substructure. As a result we can observe that if the input substructures are larger, the searching space of possible result structures is smaller. We will show that even when the transformation of a given problem into a SAT task results in a huge amount of variables, solutions can be found very quickly (seconds for 10^5 variables).

Generally, we will find no solution if the problem is overconstrained, one or a few solutions if the constraints are defined properly, or a large number of solutions if the problem is underconstrained. In this paper, we focused on situations when the result structure is almost unique. If we detect more solutions, we try to supply additional substructures or constraints using detected sources of ambiguities. Thus, our method might be used for controlling a process of substructure observations to fill gaps and ambiguities in the result structure.

The proposed method follows the principle of minimum description length [12] using which we replace a large number of small observations with a structure that explains or captures a majority of small observations. While in theory we could pose the search for a minimum structure, we do not insist on such a result in this paper. Generally, such

theoretically set-up problems lead to NP problems. However, in many practical examples we are able to find supplementary information, such as additional attributes. As mentioned above, we also assume that observed data are not provided as standalone atomic constraints between two logical variables but as small chunks of structures. If such additional information is supplemented with a suitable observation procedure we can often find a polynomial solution [1].

In this paper we consider a treatment of the actual algorithm complexity beyond the scope of this paper. Also, we do not treat a number of other possible aspects in depth, such as missing information or speculations on ambiguous parts of the result structure. We will just demonstrate a principle of the method. Also we will demonstrate that even when small instances of a problem lead to a rather huge number of variables used by the underlying SAT solver, processing times remain feasibly short.

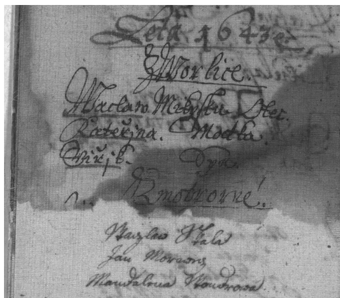
In the following sections we will introduce the Alloy tool, a bounded model checker. We will show how such observed substructures and related constraints for the result structure can be represented using the Alloy language. As a model checker, Alloy is mostly used for verifications of models represented as collections of constraints. The solver takes the constraints of a model and finds structures that satisfy them. It can be used both to explore the model by generating sample structures, and to check properties of the model by generating counterexamples [7].The simulation and exploration of model properties through the generating of samples is the feature we exploit in our proposed method. We specify the observed substructures and constraints that must be satisfied by the resulting structure and constraints determining how the input substructures limit the result structure as a problem input. The model expressed in relational logic is translated by Alloy into a corresponding boolean logic formula and an off-the-shelf SAT-solver is invoked on the boolean formula. In the event that the solver finds a solution, the result is translated back into a corresponding binding of constants to variables in the relational logic model [14, 15].

We should note that the art of model specification for our problem resides in constraining possible result structures in such a way that does not produce unwanted speculations. The task of any model checker is to search for any solution that satisfies the constraints. That means, if model constraints enable the addition of extra entities into the result structure and the related part of the structure is underconstrained by the input substructures, then the model checker generates structures that might not be valid. In fact, the same situation happens in science generally. Having some observations so far, we created a theory that covers these observations. If we find additional facts that are contradictory with the theory, we will try to adjust the theory to cover the new facts. In a similar way, if the result structure found by our method seems to be suspicious we need either to add more constraints to remove such speculations or search for additional substructure observations that would disambiguate the issue.

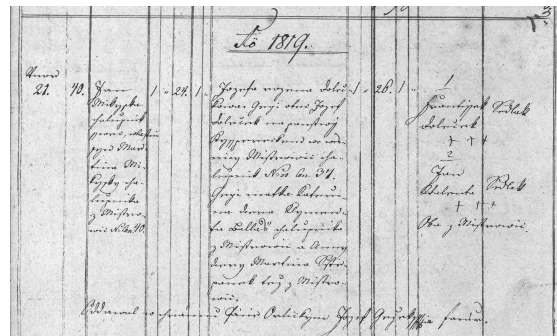
To demonstrate our proposed method, we selected a simplified domain of genealogy. We will search for a family tree given observation facts such as birth events or marriage events. However, we provide some highlights of the Alloy language in the following section before concrete steps of the proposed method are described.

3 Exemplar Problem from Genealogy Domain

The basic task in the domain of genealogy is the composition of a family tree from scattered facts about relatives. Of course, a family tree might be extended in a number of possible ways. We will simplify significantly the real problem to demonstrate fundamental properties of our method.



(a) A record in Czech from 1643 on the birth of Jirik Mikiska. His fathers name was Vaclav Mikiska and his mothers name Katerina. He was born in the village Orlice.



(b) A marriage event from 1819. Groom Jan Mikyska, son of Martin Mikyska from Mistrovice n. 40, married bride Josefa, born Doleckova, daughter of Josef Dolecek from Mistrovice n. 37 and Katerina, daughter of Klyment Balas from Mistrovice and Anna, daughter of Martin Stepanek from Mistrovice

Fig. 1: Genealogy Events

Basic facts about relatives cover the following three events. A birth event describes the birth of a person. The record of such an event often contains the following items: the name of the child, the names of its father and mother, the place of birth, and the date of birth. Information on grandparents is usually also provided. A marriage event describes a situation when two people are married. Again, the record of a marriage event contains items such as the date, the name of the groom, the name of the bride, the place of the marriage, a description of parents of the groom and the bride. The third event that provides significant information on people is a death event. The record provides information on when, where, and who died, often at what age and what the cause of the death was.

Facts about events can be discovered through discussions with people for the last 100-150 years. Facts on earlier events might be found in parish registers. We should note that records or their parts might not be readable or are missing because of fire, flood, mold, special contemporary styles of handwriting, etc. The amount of detail in records differs significantly.

Examples of two such records are depicted on Figure 1a and Figure 1b. After a short training in reading the contemporary handwriting, one can find that the record

in Figure 1a describes the birth of Jirik Mikiska. The following record from 1819 in Figure 1b contains much more information. It is about the marriage of Jan Mikyska and Josefa Doleckova. To simplify our example we drop information on dates and places. Also we simplify the treatment of names. We will use only simple single short labels instead of full names.

4 Basics of Alloy

Alloy is a formal notation based on relational logic [7, 2]. In this section, we only introduce the key characteristics of the notation as they are applied in our family tree example.

```
sig ASet {}
abstract sig Sex {}
one sig Male, Female extends Sex {}
```

An empty signature is used to introduce the concept of `ASet`. A *signature* introduces a set of atoms. A set can be a subset of another set. For example, `Male` and `Female` are (disjoint) extensions of `Sex`. An *abstract* signature, such as `Sex` has no elements except those belonging to its extensions. A multiplicity keyword, such as `one`, `lone`, `some`, `set`, placed before a signature declaration constrains the number of elements in the signatures set. Declaring an abstract signature with scalar extensions (`one`) introduces an enumeration, e.g. , `Male` and `Female`. In our method one often needs to restrict the elements of sets to be explicitly defined elements to avoid generating unwanted speculative entities. The enumeration technique is suitable for such constraints.

In our family tree example we will need to encode input substructures and the entities of result structure of family tree. We will label **the input signatures as events** and we will need only two such events: `BirthEvent` recording attributes of a child birth and `MarriageEvent` describing an event if a man and a woman get married. We will need two main signatures for family tree coding, i.e. `Person` describing composed knowledge on a given person, with its two extensions `Man` and `Woman`, and `Marriage` with collected information about a married couple of people.

```
abstract sig Name {}
abstract sig BirthEvent {childName: Name, childSex: Sex,
  fatherName: lone Name, motherName: lone Name,
  childPerson: Person, childParents: Marriage}
abstract sig MarriageEvent {groomName: lone Name, brideName: lone Name,
  groomFatherName: lone Name, brideFatherName: lone Name, marriage: Marriage}
```

Relations are declared as fields in signatures. For example, fields `childName`, `childSex`, `fatherName`, `motherName` introduce relations whose domain is `BirthEvent`. The range of a given relation is specified by the expression after the colon, in our example often as a multiplicity symbol and a signature (default multiplicity is `one`). Elements of relations are n-ary tuples. Scalars are also relations, i.e. singletons. The quintessential relational operator is *composition*, or *dot join*. To join two tuples $p = \{(N0, A0)\}$ and $q = \{(A0, D0)\}$ written as $p.q$ one first check whether the last atom of the first tuple matches the first atom of the second tuple. If not, the result is empty. If so, it is the tuple that starts with the atoms of the first tuple, and finishes with the

atoms of the second, omitting just the matching atom, e.g. $\{(N0, A0)\}.\{(A0, D0)\} = \{(N0, D0)\}$.

```
abstract sig Person {personSex: Sex, personName: Name,
  personFather: lone Man, personMother: lone Woman, personParents: lone Marriage}
sig Man extends Person {}{personSex = Male}
sig Woman extends Person{}{personSex=Female}
sig Marriage {husband: lone Man, wife: lone Woman, children: set Person}
fact MarriageBasic1 {personFather = personParents.husband
  personMother = personParents.wife}
```

We introduce the first signature of the **result structure fragment**, `Person`. Besides the relations `personSex` determining the sex of the person and `personName` coding his/her name, signature `Person` provides three relations providing binding with other people, such as mother and father, and their marriage. Signature `Marriage` binds information of married people and collects links to their own children. Constraints that are assumed always to hold are recorded as *facts*. SAT method ensures validity of facts in any direction of relations. Facts might be recorded as *signature facts* as exemplified in extensions `Man` and `Woman` with the constraints immediately following their signatures. For example, we require that any tuple of relation `personSex` describing a man has a value `Male`. Then, facts can be introduced in any order as a paragraph of its own, labelled by the keyword *fact* and consisting of a collection of constraints. In fact, both relations `personFather` and `personMother` are redundant, here introduced for demonstration purposes. The fact `MarriageBasic1` requires that relation `personFather` is one to one mapping with regard to dot join composition `personParents.husband`; similarly `personParents.wife` and `personMother`.

Alloy also provides operators for sets, such as intersection (`&`), union (+), subset (`in`), equality (=), and operators for relations, such as transpose (˜), transitive closure (ˆ), reflexive-transitive closure (`*`), identity (`ident`). Standard logical operators are also provided, e.g. negation (`not` or (`!`),), conjuction (`and`), disjunction (`or`), implication (+).

In this section we introduced very basic notation elements of Alloy language. In fact, we have also specified all signatures we need to describe both input substructures, e.g. our family events, and the result structure fragments of family tree. We still miss a number of constraints that must be satisfied by the output family tree and relations in which input substructures drive the form of the result family tree. This will be described in the next section.

5 Genealogy Relations in Alloy

A careful reader noticed that signatures `BirthEvent` and `MarriageEvent` define relations `childPerson`, `childParents`, and `marriage`. These relations define a basic mapping from the input substructures to the family tree. Although the elements of these relations are not provided explicitly, they serve as binding touch points through which the family tree is shaped. They could be also defined outside these event signatures. To make expressions shorter, we placed these relations into the event signatures.

```
fact MarriageBasic2 {
  no p: Person {p in p.^(personFather+personMother)} //1
  no wife & husband.*(personParents.(husband+wife)).personMother //2
  no husband & wife.*(personParents.(husband+wife)).personFather //3
  no wife.personParents & husband.personParents //4
  all pChild: Person | no children.pChild or (
    one children.pChild and pChild in pChild.personParents.children)} //5
```

We still need to define constraints valid for any family tree, then we need to define other constraints relating the input substructures to the result family tree. Fact `MarriageBasic2` sets additional constraints. Constraint 1 ensures nobody can be her/his own father or mother. Constraints 2 and 3 and 4 specify the social convention that there cannot be a marriage between blood relatives. Constraint 5 ensures consistency between relation `personParent` that binds a given child with its parents and relation `children` that collects references to all children of a given married couple.

```
fact FamilyTreeMapping1 {
  no personName.UnknownN //6
  all n: Name {not n=UnknownN => some Name2Person[n]}//7
  childPerson.~childPerson in iden //8
  ~childPerson.childPerson in iden //9
  childParents.~childParents in iden //10
  ~childParents.childParents in iden //11
  all p:Person { (no childPerson.p and
   (marriage.husband.p).groomFatherName = UnknownN) => no p.personParents}//12
  all m: Marriage | some childParents.m or some marriage.m  or
    some marriage.husband.personParents.m //13
  all be: BirthEvent {be.childPerson in be.childParents.children}} //14
```

We need also to constrain the amount of entities that can be generated in the family tree. The model checker generates general solutions, i.e. also those which have no justification in observations. Thus, we need to relate generated entities of the family tree to the provided observations. So, constraint 6 states no `Person` is generated for unspecified names. However, any provided `Name` will be reflected by some `Person` instances by constraint 7. All `BirthEvent` events will be reflected by exactly one child `Person` and one instance of `Marriage` using constraints 8, 9, 10, and 11. By constraint 12 no `Marriage` instance is generated if there is not birth event or marriage event evidence. Also all `Marriage` instances must be supported by a birth event or by a married couple or by their grandparents through constraint 13. Constraint 14 ensures propagation of `BirthEvent` into the family tree.

```
fact FamilyTreeMapping2 {
  all be: BirthEvent | one pChild: Person {
    (pChild = be.childPerson) and (pChild.personName = be.childName) and
    (pChild.personSex    = be.childSex) and
    ((be.motherName != UnknownN or be.fatherName != UnknownN) =>
      pChild.personParents = be.childParents)
    and (be.motherName  != UnknownN => (
      (Mother[pChild].personName = be.motherName) and
      (Mother[pChild].personSex = Female))) and
    (be.motherName  = UnknownN => no pChild.personMother)
    and (be.fatherName  != UnknownN => (
      (Father[pChild].personName = be.fatherName) and
      (Father[pChild].personSex = Male))) and
    (be.fatherName  = UnknownN => no pChild.personFather)}
```

Fact `FamilyTreeMapping2` propagates information from `BirthEvent` instances into `Person` and `Marriage` instances. The propagation is performed conditionally only in situations when names of people are provided already.

```
fact FamilyTreeMapping3 {
  all me: MarriageEvent | one m: Marriage {
    (m = me.marriage) and (me.groomName  != UnknownN =>
      me.groomName = m.husband.personName) and
    (me.brideName  != UnknownN  => me.brideName   = m.wife.personName) and
    (me.groomFatherName != UnknownN =>
      m.husband.personParents.husband.personName = me.groomFatherName)}
  all m1: Marriage | all m2: Marriage
    {m1.husband = m2.husband and m1.wife = m2.wife => m1=m2}}
```

Similarly, `FamilyTreeMapping3` propagates information from `MarriageEvent`. It follows the same strategy as the previous fact. Finally, we need to provide the input data. As a demonstration we select a very simple set of input substructures. There are 7 names, three instances of `BirthEvent`, including those with unspecified names, and one instance of `MarriageEvent` with richer details.

```
one sig UnknownN, N1, N2, N3, N4, N5, N6, N7 extends Name {}
one sig N2_BE extends BirthEvent{}{childName=N2 childSex=Female
  fatherName=N5 motherName=N1}
one sig N3_BE extends BirthEvent{}{childName=N3 childSex=Female
  fatherName=N6 motherName=N1}
one sig N4_BE extends BirthEvent{}{childName=N4 childSex=Male
  fatherName = UnknownN motherName=N2}
one sig M6_1 extends MarriageEvent {}{groomName=N6 brideName=N1
  groomFatherName=N7 brideFatherName=UnknownN}
```

6 Experiments and Discussion

A simple command of Alloy `run {} for 7 Person, 7 Marriage` can trigger a generation of the family tree. The maximum number of people and `Marriage` instances must be either estimated (the maximum number of people is determined by the number of fields with names, also the number of marriages is constrained by the number of birth and marriage events). The result family tree is depicted on Figure 2. The solution was found using 16978 vars, 544 primary vars, 40721 clauses in 109 ms using Sat4j solver. Characteristics of other experiments with a different number of input events are collected in Table 1.

Table 1: Performance of the proposed method with different problem size.

the number of input birth events	3	9	9	12	12
the number of input marriage events	1	1	4	1	6
the number of people	7	17	17	21	26
the number of marriages	4	9	8	13	13
the number of variables	16978	221736	233740	588927	614633
the number of primary variables	544	3008	3324	5638	6256
the number of clauses	40721	541951	580071	1421000	1501564
processing time in milliseconds	109	4949	21338	48516	27846

From Table 1 we cannot draw any conclusions regarding additional constraints provided by input substructures like marriage event. In our experiments, these additional

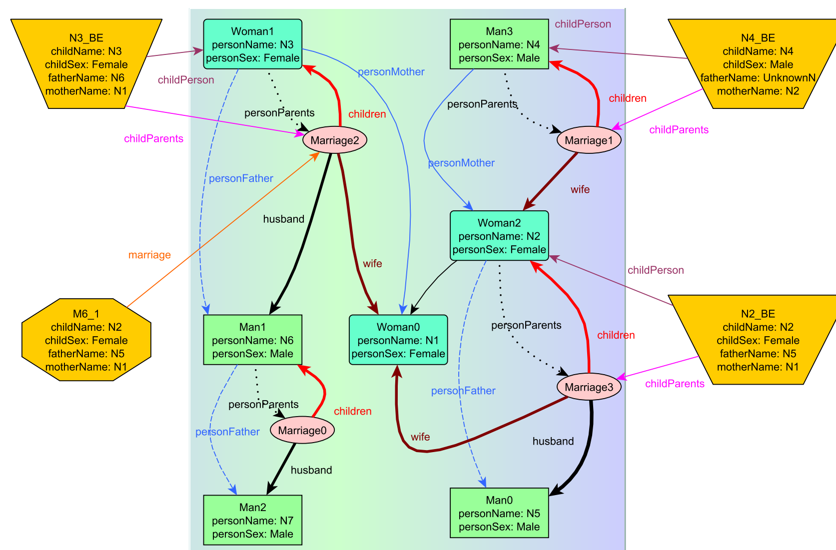


Fig. 2: The result family tree is synthetized by the constraints and the provided input substructures of birth and marriage events. The diagram makes explicit feeding of the family tree from the input events shown outside the box of family tree. The diagram generated by the tool was laid out manually.

events were used mainly as confirmations of some relations inside of the result family tree. They meant adding a relatively small number of other variables, but they resulted in rather different processing time, both longer and shorter. Our original speculation was that such additional input constraints should shorten processing time as the search space is more constrained.

7 Conclusion

In this paper we demonstrated that using a set of relational substructures observed as partial views on an unknown system we can compose the entire structure. The proposed method utilizes a transformation from relational specification into SAT problem. We use Alloy Analyzer that provides a suitable relational notation language and bounded model checking. Constraints must restrict both properties of the result structure and how input information is propagated into the result structure. The proposed method was presented on exemplar case taken from the genealogy domain. Using a few constraints we are able to find the underlying family tree. Although the number of variables of the related SAT problem is huge as expected, the search for the solution is fast because observation facts of overlapping substructures are bound by unique identities of substructures. Nevertheless, some additional issues such as time, dates, a selection of suitable mappings, and sensitivity to unspecified variables need to be resolved before the method can be utilized in more practical problems.

Acknowledgments. We would like to thank the reviewers for their comments, which helped improve this paper considerably. This work was supported by the Systems for Identification and Processing of Signaling and Transmission Protocols VF20132015029 project.

References

[1] Angluin, D.: Learning regular sets from queries and counterexamples. Information and Computation 75(2), 87–106 (1987)

[2] Baresi, L., Spoletini, P.: On the use of Alloy to analyze graph transformation systems. In: Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G. (eds.) Graph Transformations, Lecture Notes in Computer Science, vol. 4178, pp. 306–320. Springer Berlin Heidelberg (2006)

[3] Du, N., Wu, B., Pei, X., Wang, B., Xu, L.: Community detection in large-scale social networks. In: Proceedings of the 9th WebKDD and 1st SNA-KDD 2007 Workshop on Web Mining and Social Network Analysis. pp. 16–25. WebKDD/SNA-KDD '07, ACM, New York, NY, USA (2007)

[4] Duda, R.E., Hart, P.E.: Pattern Classification and Scene Analysis. John Wiley (1973)

[5] Duda, R.O., Hart, P.E., Stork, D.G.: Pattern Classification. Wiley, New York, 2 edn. (2001)

[6] Fukunaga, K.: Introduction to Statististical Pattern Recognition. Academic Press, second edn. (1990)

[7] Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press, ISBN 978-0-262-10114-1 (2006)

[8] Muggleton, S., de Raedt, L.: Inductive logic programming: Theory and methods. The Journal of Logic Programming 19-20(0), 629 – 679 (1994)

[9] Newman, M.: Networks: an introduction. Oxford University Press, Inc. (2010)

[10] Popper, K.R.: Logika vedeckeho zkoumani (Logik der Forschung, German original 1934). OIKOYMENH (1997)

[11] Raedt, L.D.: Logical and Relational Learning. Springer (September 2008)

[12] Rissanen, J.: Modeling by shortest data description. Automatica 14(5), 465–471 (Sep 1978)

[13] Russell, S.J., Norvig, P.: Artificial Intelligence, A Modern Approach. Pre, third edn. (2010)

[14] Torlak, E., Dennis, G.: Kodkod for alloy users. In: First Alloy Workshop, colocated with the Fourteenth ACM SIGSOFT Symposium on Foundations of Software Engineering. Portland, Oregon (November 6 2006)

[15] Torlak, E., Jackson, D.: Kodkod: A relational model finder. In: Grumberg, O., Huth, M. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science, vol. 4424, pp. 632–647. Springer Berlin Heidelberg (2007)

[16] Uszkoreit, H.: Methods and applications for relation detection potential and limitations of automatic learning in ie. In: Proceedings of International Conference on Natural Language Processing and Knowledge Engineering, NLP-KE 2007. pp. 6–10. Beijing, China (Aug 30-Sep 1 2007)

[17] Welty, C., Fan, J., Gondek, D., Schlaikjer, A.: Large scale relation detection. In: Proceedings of the NAACL HLT 2010 First International Workshop on Formalisms and Methodology for Learning by Reading. pp. 24–33. Association for Computational Linguistics, Los Angeles, California (Jun 2010)